

PGI[®] Compilers for Cray XT5 Systems

Dave Norton
The Portland Group
Dave.Norton@pgroup.com

Craig Toepfer
The Portland Group
Craig.Toepfer@pgroup.com

ORNL User Group Conference
10 March 11



- C, C++, F2003 compilers
- Optimizing, Vectorizing, Parallelizing
- Graphical parallel debugger, profiler
- AMD & Intel, 32 & 64-bit, SSE & AVX
- PGI Unified Binary™ technology
- Linux, MacOS, Windows
- Visual Studio integration on Windows
- CUDA Fortran for NVIDIA GPUs
- CUDA C for both X86 and NVIDIA
- PGI Accelerator™ Programming Model



www.pgroup.com

The Portland Group®

PGI Milestones

- **1989 PGI Formed**
- 1991 Pipelining i860 Compiler
- Intel Paragon support
- 1994 Parallel i860 Compiler
- 1996 ASCI Red TFLOPS Compiler
- 1997 Linux/x86 Compiler
- 1998 OpenMP for Linux/x86
- 1999 SSE/SIMD Auto-vectorization
- 2001 VLIW ST100 Compiler
- 2003 64-bit Linux/x86 Compiler
- 2004 ASCI Red Storm Compiler
- 2005 PGI Unified Binary
- 2006 PGI Visual Fortran
- 2007 64-bit MacOS Compiler
- **2008 PGI Accelerator Compiler**
- **2009 CUDA Fortran**
- **2011 CUDA C for x64 and MIC**

PGI[®] 2011 Features

- ❑ PGI Accelerator[™] Programming Model
 - High-level, Portable, Directive-based Fortran & C extensions (no C++, yet)
 - Supported on NVIDIA CUDA GPUs
- ❑ PGI CUDA Fortran
 - Extended PGI Fortran, co-defined by PGI and NVIDIA
 - Lower-level explicit NVIDIA CUDA GPU programming
- ❑ **PGI CUDA C for x64 and MIC**
 - Demonstration of CUDA C for x64 at SC 2010 in New Orleans
 - Mixing of CUDA C, CUDA Fortran and PGI Accelerator directives
- ❑ Compiler Enhancements
 - **F2003 – object oriented features**
 - Latest EDG 4.1 C++ front-end – more g++/VC++ compatible, zero cost exception handling (-zc_eh)
 - **AVX code generation, code generator tuning**
- ❑ PGPROF Enhancements
 - Uniform performance profiling across Linux, MacOS and Windows
 - **x64+GPU performance profiling**
 - Updated Graphical User Interface (GUI)

Compiling codes with PGI

PGI is the default compiler on the XT5 systems.

Cray supplies wrappers to all of the compilers on the system so that the Fortran compiler is always invoked as “ftn”, the C compiler as “cc”, and C++ as “CC” regardless of the actual compile vendor being used.

```
> module list
```

```
pgi/11.2.0
```

```
> ftn -V foo.f -o foo
```

```
pgfortran 11.2-0 64-bit target on Linux -tp barcelona-64
```

Or you can call the compiler directly with “*pgfortran*” but you won’t get the Cray library wrappers for use in the XT5 system

Using a different version of PGI

On the Cray, to change the version of the PGI compiler, you need to switch modules:

```
> module switch pgi/11.2.0 pgi/10.2.0
```

```
> ftn -V foo.f -o foo
```

```
pgf90 10.2-0 64-bit target on Linux -tp barcelona
```

On your workstation, if you have multiple versions of PGI installed, you can invoke a different version of the compiler through the compile driver:

```
> pgfortran -V10.2 hello.f -o hello
```

```
pgfortran 10.2-0 64-bit target on Linux -tp istanbul-64
```

Changing target processors

The PGI compile driver by default compiles for the processor on which the compilation takes place. The driver allows you to easily cross compile for another target processor:

```
> pgfortran -V foo.f -o foo -tp istanbul-64
```

```
pgfortran 11.2-0 64-bit target on Linux -tp istanbul-64
```

At the request of Cray users at ORNL - new to 11.2 – you can set the target processor in your ~/.mypgirc or the siterc file. This can then be overridden with the –tp flag on the command line (Cray still needs to change compile scrips):

```
set PREOPTIONS=-tp=barcelona-64
```

Then override on the commandline:

```
> pgfortran -V foo.f -o foo -tp istanbul-64  
pgfortran 11.2-0 64-bit target on Linux -tp istanbul-64
```

Changing target processors

Why change the target processor?

You intend to run on a processor different then the default. (On JaguarPF, you are actually cross compiling for the istanbul-64 processor)

The target system has more then one CPU type on it. Suppose JaguarEF contains both AMD and Intel CPUs.

```
> ftn -V -tp sandybridge-64,bulldozer-64
```

```
pgfortran 11.2-0 64-bit target on Linux -tp sandybridge-64,bulldozer-64
```

The creates a **PGI Unified Binary** which contains optimized code for the sandybridge chip and optimized code for the bulldozer chip in the same binary. At run time, the system determines the processor it is running on and selects that code branch, thus allowing a single executable to run on an heterogeneous target architecture.

I'm in a hurry – how do I start?

OK, OK – I have the best compilers on the world's best computer. What should I do to get my code compiled and start computing?

Recommended optimization

```
> ftn -fast foo.f -o foo
```

Invoking the compiler with the `-fast` (or `-fastsse`) flag sets common optimizations which include:

- O2
- Munroll=c:1
- Mnoframe (gives the compiler another register)
- Mlre
- Mautoinline
- Mvect=sse <= this is the vectorizer
- Mscalarsse
- Mcache_align
- Mflushz
- Mpre

Can PGI help profile my code?

Yes! PGI recommends that you use the **-Minfo=ccff** flag for all compiles.

-Minfo instructs the compiler to print out informative messages during the compilation stage.

-Minfo=ccff instructs the compiler to also imbed these informative messages in the executable itself. When you use the profiler, you can then coordinate performance information with source code lines and the compiler messages emitted when compiling that line of code.

Profiling code

Cray provides some excellent tools for profiling using hardware counters.

PGI also provides some excellent tools for profiling of code. The simplest method is to use **pgcollect**. No special build process is needed, although compiling with **-Minfo=ccff** may provide useful feedback. This imbeds the **-Minfo** messages into the executable which can then be viewed with the performance profile.

Run your code as:

```
> pgcollect a.out
```

Then view the results with the GUI tool - pgprof

```
> pgprof -exe a.out
```

Profiling code

To get a general profile for an MPI code, you may wish to just profile one of the MPI processes. Instead of launching the executable via `mpiexec`, launch a script which launches the executable instead:

```
> mpiexec -np 2000 ./doit
```

The “doit” script for code compiled and linked with MPICH2 might look like the following:

```
#!/bin/csh

if ($PMI_RANK == 0) then
    pgcollect ./test
else
    ./test
endif
```

After the run is complete, there will be only one `pgprof.out` file which can be viewed using:

```
> pgprof -exe ./test pgprof.out
```

Can the runtime help catch errors?

Yes! The PGI Fortran 11.0 compiler supports Fortran 2003.

Fortran 2003 requires that the Fortran STOP message signal all IEEE exceptions.

By default Cray will disable this capability so you don't get messages from each and every MPI process. (export NO_STOP_MESSAGE=1)

To re-enable it: unset NO_STOP_MESSAGE

You will almost always see:

```
Warning: ieee_inexact is signaling
```

Of more concern are other messages like:

```
Warning: ieee_divide_by_zero is signaling
```


Scalar, vector – what's the difference?

Scalar code produces one result for each assembly language instruction

Vector code produces multiple results – depending on the vector length of the target processor – for each assembly language instruction.

<u>Processor</u>	<u>32 bit vector length</u>	<u>64 bit vector length</u>
Barcelona	4	2
Istanbul	4	2
Bulldozer	8	4
Sandybridge	8	4
MIC	16	8
NextGen	32	16

Basic levels of vector optimization

Vectorization is the key to getting the best performance out of floating point intense codes. Current processors are capable of operating on 128 bits at a time. This means they can do 2 – double precision operations or 4 – single precision operations at the same time – as long as those operations can all be described by a single instruction (i.e. a vector operation).

AVX – coming by the end of the year, increases this to 256 bit wide units

The vectorizer performs the following operations:

- Loop interchange and loop splitting

- Loop fusion

- Memory-hierarchy (cache tiling) optimizations

- Generation of SSE instructions and prefetch instructions

- Loop peeling to maximize vector alignment

- Alternate code generation

Common impediments to vector optimization

There are several common coding issues that may prevent vectorization. The programmer may have enough knowledge to provide additional information to the compiler to work around these issues.

In C and C++ the compiler may not have enough information about the pointers passed into a subroutine to be able to determine that those pointers don't overlap. (-Msafepr option or pragma or restrict keyword)

Function calls can be inlined to allow vectorization (-Minline)

Constants may be of the wrong type (-Mfcon)

Loops may be too long or too short. In both cases, additional options to the vectorizer may be successful in generating vector code.

-Msafeptr Option and Pragma

–M[no]safeptr[=all | arg | auto | dummy | local | static | global]

all All pointers are safe

arg Argument pointers are safe

local local pointers are safe

static static local pointers are safe

global global pointers are safe

#pragma [*scope*] [no]safeptr={arg | local | global | static | all},...

Where *scope* is *global*, *routine* or *loop*

Basic levels of scalar optimization

```
> ftn foo.f -o foo
```

Invoking the compiler with no flags for optimization will set the scalar optimization level to 1 if `-g` is not specified.

```
> ftn -g foo.f -o foo
```

Invoking the compiler with no flags for optimization will set the scalar optimization level to 0 if `-g` is specified.

```
> ftn -O foo.o -o foo
```

Invoking the compiler with the `-O` flag for optimization will set the scalar optimization level to 2 regardless of whether `-g` is also specified.

Optimization levels O0 through O4 perform increasing aggressive *scalar* optimizations

Which level of optimization to start?

If you are just starting with a new code, we suggest that you try a short run of the code with optimization level **-O2**.

If the answers look good, then try the same run with the **-fast** flag.

If the answers are the same as the first run, use **-fast** as the basis for further optimizations. If the answers differ, try turning of optimizations one at a time until you find the optimization that is causing the difference. You can then track down in your code where that difference occurs and determine if it can be fixed, or if the optimization needs to be left turned off.

Turning off optimizations

Optimization flags are processed on the command line in the order in which they occur. For example - to turn on all `-fast` optimizations except loop redundant elimination:

```
> ftn -fast -Mnolre foo.o -o foo
```

Most optimizations can be turned on with the syntax `-Moptimization`

Most optimizations can be turned off with the syntax `-Mnooptimization`

Optimizations and debugging

Optimizations and debugging don't always go hand in hand, however...

```
> ftn -fast -gopt foo.f -o foo
```

-gopt inserts debugging information without disabling optimizations. It is often helpful for tracking down a code bug that only appears in optimized code, or a bug that occurs far enough into a code that running the code with no optimizations takes a painful amount of time.

Generating tracebacks

Linux uses the backtrace system call to create the stacktrace when a fault or error occurs. The only requirement is to link with the -Meh_frame option:

```
> pgfortran -Meh_frame -o x x.f90
```

Then before running the program, the following environment variable is set as follows:

```
> export PGI_TERM=trace
```

Generating tracebacks

Here is a sample traceback from within the PGI runtime.
(An attempt to deallocate an allocatable array more than one time):

```
0: DEALLOCATE: memory at (nil) not allocated
./x(__hpf_abort+0x7d) [0x40bb8d]
./x(__hpf_dealloc+0xeb) [0x40b57b]
./x(MAIN_+0x217) [0x408177]
./x(main+0x40) [0x407f40]
/lib64/libc.so.6(__libc_start_main+0xf4) [0x2b877285e154]
./x [0x407e69]
```


Here is a sample traceback from a SEGV in user code:

Error: segmentation violation, address not mapped to object

```
rax 0000000005f45908, rbx 0000000000000001, rcx 00000000000187f9
rdx 000000000000187f9, rsp 00007fffcdaef9a0, rbp 00007fffcdaef9a0
rsi 00007fffcdaef9c4, rdi 00002ab2dd77e020, r8 00000000ffffffff
r9 0000000000000000, r10 0000000000000022, r11 0000000000000246
r12 0000000000000001, r13 00007fffcdaefae0, r14 0000000000000000
r15 0000000000000000
```

```
/lib64/libpthread.so.0 [0x2ab2dd1ebc10]
```

```
./y(init_+0x1f) [0x4081bf]
```

```
./y(MAIN_+0x9b) [0x407ffb]
```

```
./y(main+0x40) [0x407f40]
```

```
/lib64/libc.so.6(__libc_start_main+0xf4) [0x2ab2dd468154]
```

```
./y [0x407e69]
```

What does *this* flag do?

There are too many compiler flags to remember all of their options. You can get help in several places:

> `man pgfortran`

> `pgfortran -fast -help` – gives help on `-fast`

Full PDF manuals are online in (e.g)

/opt/pgi/11.2.0/linux86-64/2011/doc

Manuals are also available at:

<http://www.pgroup.com/resources/docs.htm>

What exactly is being optimized?

Optimization is as much a user exercise as it is a compiler exercise. To see what the compiler thinks of your code, compile using the `-Minfo` flag.

```
> pgfortran -fast -Minfo=ccff foo.f -o foo
```

Use the information generated by `-Minfo` to help identify coding issues and locate places where code can be improved so the compiler can do an optimal job on it.

```
> pgfortran -Minfo -help
```

Use `-Minfo` to see which loops vectorize

```
> ftn -fast -Mipa=fast -Minfo=ccff -S graphRoutines.f90
```

localmove:

334, Loop unrolled 1 times (completely unrolled)

343, Loop unrolled 2 times (completely unrolled)

358, Generating vector sse code for inner loop

364, Generating vector sse code for inner loop

Generating vector sse code for inner loop

392, Generating vector sse code for inner loop

423, Generating vector sse code for inner loop

Use `-Mneginfo` to see why things don't vectorize

Additional compiler optimizations

The `-fast` flag is the 90/90 solution for code optimization. That is, it achieves about 90% of the possible performance for about 90% of the codes.

That means there are some additional areas that can be explored.

Interprocedural analysis can be helpful for C codes and Fortran codes without interface blocks. (Interface blocks are to the language specification what IPA is to the compiler)

```
> ftn -fast -Minfo -Mipa=fast,inline foo.f -o foo
```

***If compiling and linking are done in separate steps, you must be sure to pass the IPA flag to the linker too.

IPA involves an additional pass of the compiler.

Additional IPA optimizations

The suggested usage for IPA is to apply `-Mipa=fast` globally

The `-Mipa` flag has a *large* number of options that may be helpful in certain circumstances. These options are generally best applied to a specific subroutine to address a specific issue.

A couple of the more interesting flags include:

`-Mipa=libopt` This allows recompiling and optimization of routines from libraries using IPA information. If you make extensive use of libraries in your code, try compiling those libraries with `-Mipa=fast` so that you have the option of using IPA when you link your application to that library

`-Mipa=safeall` This declares that all unknown procedures are safe.

Additional compiler optimizations

Several memory management options are available and may be beneficial depending on how your code accesses memory. *Smartalloc* tends to do a better job managing memory than standard Unix malloc.

Smartalloc can make use of “big pages”. Using big pages helps to minimize the number of TLB misses. This option tends to be helpful for codes that do a big initial allocate and then manage their own memory.

```
> ftn -fast -Minfo -Mipa=fast,inline -Msmartalloc=huge foo.f -o foo
```

***-Msmartalloc must be used to compile main, and also to link the program

Additional compiler optimizations

Inlining can have a significant impact on application performance. It's most dramatic effects tend to be on C++ codes which have many many small functions.

Inlining can be done at several different points in the compilation.

-Minline/autoinline - during the regular compilation phase

-Mipa=inline - during the recompile for IPA

Inline libraries - created during the “make” process

The auto inliner is for C/C++ only. This enables inlining functions with the inline attribute. The suboptions control how the auto inliner operates.

`-M[no]autoinline`

Enable inlining of functions with the inline attribute.

`-Mautoinline` is implied with the `-fast` switch. The options are:

`levels:n` Inline up to `n` levels of function calls; the default is to inline up to 10 levels.

`maxsize:n` Only inline functions with a size of `n` or less. The size roughly corresponds to the number of statements in the function, though the correspondence is not direct. The default is to inline functions with a size of 100 or less.

`totalsize:n`

Stop inlining when this function reaches a size of `n`. The default is to stop inlining when a size of 8000 has been reached.

Creating and Using Inline Libraries

Use of -Minline/-Mextract to create an inline library. This works for all languages(C/C++/FORTRAN). To create an inline library with -Mextract do the following:

```
pgfortran -Mextract=lib:libfloat.il -c add.f90
pgfortran -Mextract=lib:libfloat.il -c sub.f90
pgfortran -Mextract=lib:libfloat.il -c mul.f90
pgfortran -Mextract=lib:libfloat.il -c div.f90
```

This creates an inline library name libfloat.il which can be used during compilation as follows:

```
pgf90 -fast -Minline=libfloat.il -c -Minfo -Mneginfo
      driver.f90
```


The -Minfo messages for this compile are:

test:

```
14, Generated an alternate loop for the loop
    Generated vector sse code for the loop
21, Generated an alternate loop for the loop
    Generated vector sse code for the loop
22, add inlined, size=2, file add.f90 (2)
33, Generated an alternate loop for the loop
    Generated vector sse code for the loop
34, sub inlined, size=2, file sub.f90 (2)
45, Generated an alternate loop for the loop
    Generated vector sse code for the loop
46, mul inlined, size=2, file mul.f90 (2)
57, Generated an alternate loop for the loop
    Generated vector sse code for the loop
58, div inlined, size=2, file div.f90 (2)
```

As a result of inlining the functions add, sub, mul, and div the compiler was then able to vectorize the loops that contained those calls.

Use of -Mipa=inline to inline functions/subroutines. This works for all languages(C/C++/FORTRAN). Create the library using the -Mipa=inline option as follows:

```
pgfortran -Mipa=fast,inline -Minfo -Mneginfo -c add.f90
pgfortran -Mipa=fast,inline -Minfo -Mneginfo -c sub.f90
pgfortran -Mipa=fast,inline -Minfo -Mneginfo -c mul.f90
pgfortran -Mipa=fast,inline -Minfo -Mneginfo -c div.f90
```

```
ar cr libfloat.a add.o sub.o mul.o div.o
```

This creates a library named libfloat.a which can be used during compilation as follows(need to use the libinline suboption):

```
pgf90 -fast -Mipa=fast,inline,libinline -c -Minfo -Mneginfo
      driver.f90
pgf90 -fast -Mipa=fast,inline,libinline -o d driver.o
      libfloat.a
```

The -Minfo messages for this compile are:

test:

14, Generated an alternate loop for the loop
Generated vector sse code for the loop
21, Loop not vectorized/parallelized: contains call
33, Loop not vectorized/parallelized: contains call
45, Loop not vectorized/parallelized: contains call
57, Loop not vectorized/parallelized: contains call

IPA: Recompiling driver.o: stale object file

test:

0, Pointer c is only set via allocate statements
Pointer b is only set via allocate statements
Pointer a is only set via allocate statements
Function add does not write to any of its arguments
Function add does not reallocate any of its arguments
Function add does not reassociate any of its pointer arguments
Function add does not reallocate any global variables
Function add does not reassociate any global pointers
Function add does not read any global (common/module) variables
Function add does not write any global (common/module) variables
Function sub does not write to any of its arguments
Function sub does not reallocate any of its arguments
Function sub does not reassociate any of its pointer arguments
Function sub does not reallocate any global variables
Function sub does not reassociate any global pointers
Function sub does not read any global (common/module) variables
Function sub does not write any global (common/module) variables
Function mul does not write to any of its arguments

Compiler optimizations and accuracy

There are a number of compiler options that offer the possibility of significant performance improvement at the expense of accuracy. If you are having numerical issues, you might tighten some restrictions.

- Kieee** – floating point strictly conforms to IEEE 754 standard. (off by default)
- Ktrap** – turns on the behavior of the processor when exceptions occur
- Mdaz** – mode to treat IEEE denormalized input numbers as zero
- Mflushz** – set SSE to flush-to-zero mode (on with **-fast**)
- Mfprelaxed** - perform certain floating point operations using relaxed precision when it improves the speed. (This is the default mode on most other vendor's compilers)

Using more then one core

There are three general techniques for using more then one core for a computation. Of course, on large XT5 machines, all codes implement parallelism through MPI.

While most codes are MPI everywhere, some codes benefit by using the shared memory on the node through either automagic parallelizing by the compiler or/and OpenMP. OpenMP compilation is invoked with the `-mp` flag, automagic parallelization with the `-Mconcur` flag.

Environment variables which can effect OpenMP performance include:

`OMP_SCHEDULE` – can be static, dynamic, guided or auto

`OMP_NUM_THREADS` – specifies the number of threads to use

`OMP_STACKSIZE` – override the default stack size for new threads.

Explicit Function Inlining

–Minline[=[lib:]<inlib> | [name:]<func> | except:<func> |
size:<n> | levels:<n>]

[lib:]<inlib> Inline extracted functions from *inlib*

[name:]<func> Inline function func

except:<func> Do not inline function func

size:<n> Inline only functions smaller than n
statements (approximate)

levels:<n> Inline n levels of functions

***For C++ Codes, PGI Recommends IPA-based
inlining or –Minline=levels:10!***

SMP Parallelization

- ❑ **–Mconcur for auto-parallelization on multi-core**

Compiler strives for parallel outer loops, vector SSE inner loops

–Mconcur=innermost forces a vector/parallel innermost loop

–Mconcur=cncall enables parallelization of loops with calls

- **–mp to enable OpenMP parallel programming model**

OpenMP programs compiled w/out –mp “just work”

Starting in 7.0, two options for idle policy

- ❑ **–Mconcur and –mp can be used together!**

Miscellaneous Optimizations (1)

- ❑ **-Mfprelaxed** – single-precision sqrt, rsqrt, div performed using reduced-precision reciprocal approximation
- ❑ **-lacml** and **-lacml_mp** – link in the AMD Core Math Library
- ❑ **-Mprefetch=d:<p>,n:<q>** – control prefetching distance, max number of prefetch instructions per loop
- ❑ **-tp k8-32** – can result in big performance win on some C/C++ codes that don't require > 2GB addressing; pointer and long data become 32-bits

Fortran 2003 Features in Current PGI Compiler Release

IEEE_EXCEPTIONS module

Extensions

ISO_C_Binding

c_associated

Interface procedure

move_alloc()

volatile attribute and stmt

Access to environment

Optional Kind to Intrinsic

PENDING specifier for INQUIRE

POS specifier for INQUIRE

Allow NAMELIST w/internal file

Classes

type uses CONTAINS declaration

EXTENDS_TYPE_OF intrinsic SAME_TYPE_AS intrinsic

IEEE_ARITHMETIC module Allocatable Array

c_f_pointer

Enumerators

Pass and Nopass Attribute

Pointer Reshaping

IMPORT statement

Length of names and statements

Asynchronous I/O'

IOSTAT kind in all i/o stmts

IEEE_ARITHMETIC large arrays

Type Extension(not polymorphic)

c_f_procpointer

Procedure Pointers

allocatable scalars

Square brackets

iso_fortran_env module

Wait Statement

Access = 'stream'

SIZE kind in read/write stmts

polymorphic entities

Inheritance

Typed allocation

Fortran 2003 Features in Current PGI Compiler Release

- READ blank specifier READ pad specifier WRITE delim specifier
- NEW_LINE intrinsic IS_IOSTAT_END intrinsic IS_IOSTAT_EOR intrinsic
- SYSTEM_CLOCK COUNT_RATE is real abstract interfaces
- Type-bound procedures PASS attribute NOPASS attribute
- NON_OVERRIDABLE attribute PRIVATE and PUBLIC attributes
- PRIVATE statement for type bound procedures deferred type-bound procedures
- ABSTRACT types i/o keyword encoding
- Decimal comma for i/o, dc, dp ASYNCHRONOUS attribute and stmt
- IEEE_FEATURES module Max, Min take character
- errmsg on allocate/deallocate Mixed component accessibility
- Sourced allocation (non-polymorphic) Associate Construct
- Sourced allocation (polymorphic types)

Fortran 2003 Remaining Features

	Compiler Release
• deferred-character-length	11.0
• generic & derived type the same	11.0
• sourced allocation (deferred character)	11.0
• PROTECTED attribute and stmt	11.0
• stop stmt warns about FP exc.	11.0
• rename user-defined operators	11.0
• array constructor syntax	11.0
• structure constructors	11.0
• SELECTED_CHAR_KIND intrinsic	11.0

Fortran 2003 Object Oriented Features

	Compiler Release
• generic type-bound procedures	11.0
• select type construct	11.0
• unlimited polymorphic entities	11.0
• typed allocation for unlimited polymorphic entities	11.0
• sourced allocation for unlimited polymorphic entities	11.0
• select type construct for unlimited polymorphic entities	11.0
• deferred type parameters (requires MRC 15.2 & MRC 16.2.1)	11.8
• sourced allocation (polymorphic source type with allocatable members)	11.4
• parameterized derived types (MRC 16.2.1)	11.7
• final procedures	11.5

Fortran 2003 I/O Features

Compiler Release

- | | | |
|--|------|------|
| • i/o of inf and nan (fs#3962) | 11.0 | |
| • round i/o specifier, ru,rd,etc. (writes only) | | 11.0 |
| • non-default derived type I/O | 11.x | |
| • non-default derived type I/O (type-bound procedures) | 11.x | |
| • recursive I/O w/external file | 11.x | |
| • recursive I/O w/internal file | 11.x | |
| • SIGN= Specifier | 11.0 | |
| • NEXTREC, NUMBER, RECL, SIZE kind | 11.0 | |
| • DECIMAL in INQUIRE stmt | 11.0 | |
| • F2003 NAMELIST group entities | 11.0 | |

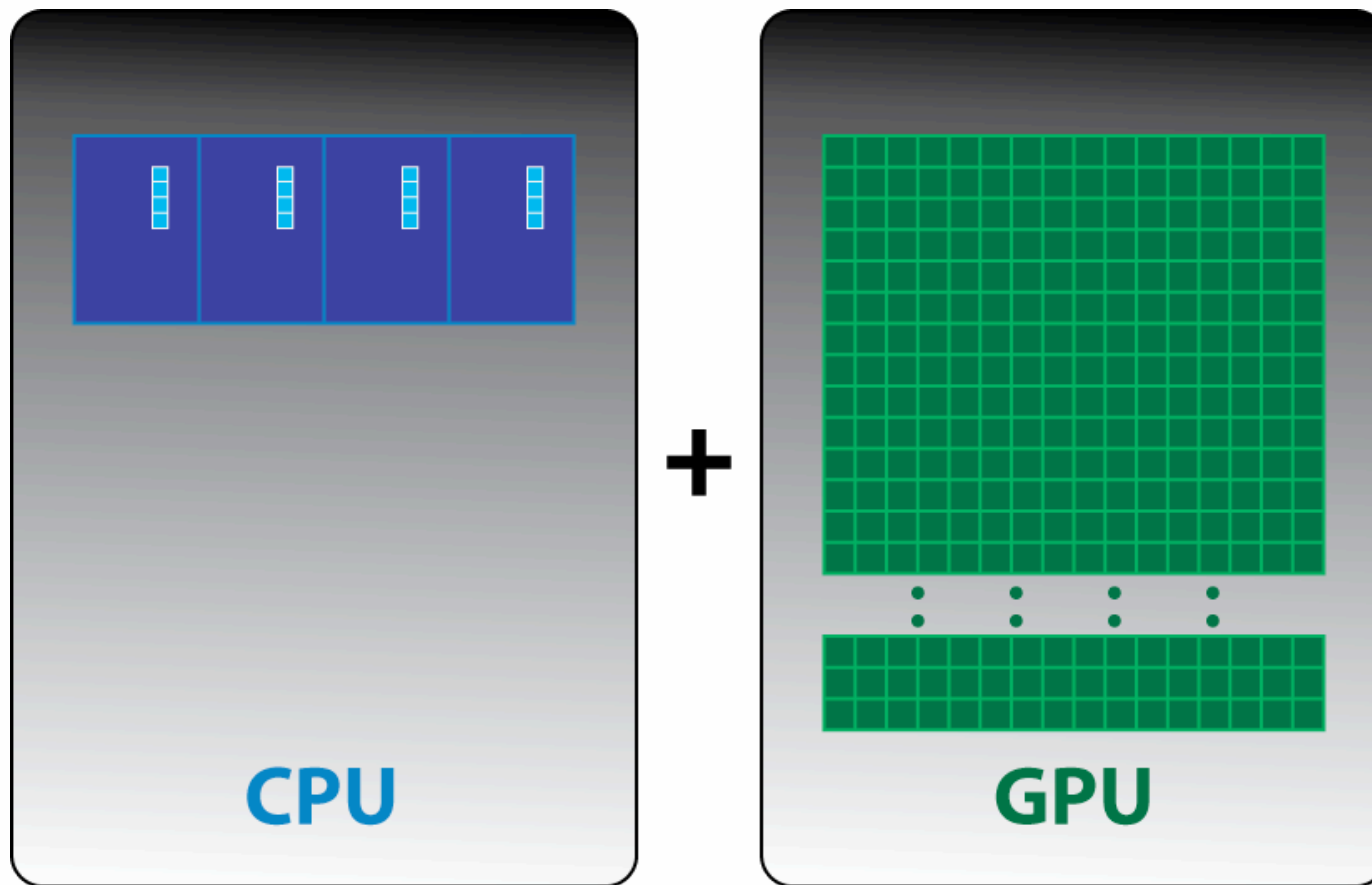
AVX Support in 11.0

- The next generation of processors from both Intel and AMD will support AVX instructions.
- AVX doubles the width of the floating point registers to 256 bits and adds 3 operand instructions resulting in more than a 2X decrease in assembly language instructions in performance critical sections of code
- AVX are *vector* instructions where one instruction operates on 8 sp, or 4 dp words at the same time, effectively doubling the performance of the CPU.
- Codes should be compiled with `-fast` for vectorization and `-Minfo` to get compiler feedback
- PGI compiled codes can make use of the Intel AVX simulator

PGI CUDA C for Multi-core x86

- ☐ **Will track NVIDIA's definition and evolution of the CUDA C language for GPUs moving forward**
- ☐ **Implementation will proceed in phases**
 - Phase 1 prototype demonstration at SC10 in New Orleans (November)
 - Phase 2 first production release in Q2 2011 with most CUDA C functionality; not a performance release
 - Phase 3 performance release in Q4 2011 leveraging multi-core and SSE/AVX to implement low-overhead native parallel/SIMD execution
- ☐ **Will eventually support execution of Device kernels on NVIDIA CUDA-enabled GPUs as well**
- ☐ **PGI Unified Binary technology will enable one binary that uses NVIDIA GPUs when present or defaults to multi-core x86 if no GPU is present**

Support x86+GPU Architecture as an Integrated Parallel System



Multicore + SSE/AVX

Massively Parallel



PGI CUDA Compilers for Multi-core x86 & NVIDIA GPUs

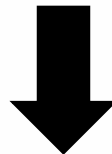
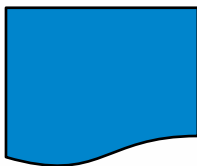
PGI CUDA C/Fortran



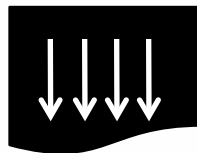
Optimization / Parallelization



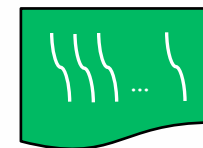
Optimized
Host Code



Parallel
Multi-core Kernels



Massively Parallel
GPU Kernels



CUDA C for GPUs vs Multi-core x86

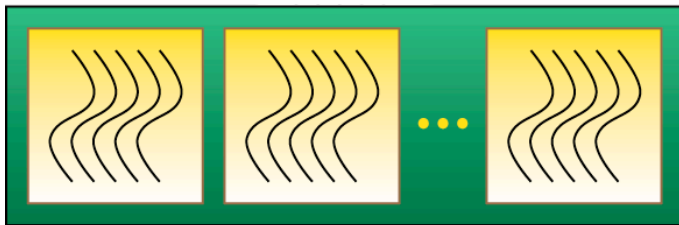
Software



Thread



Thread Block



Thread Grid

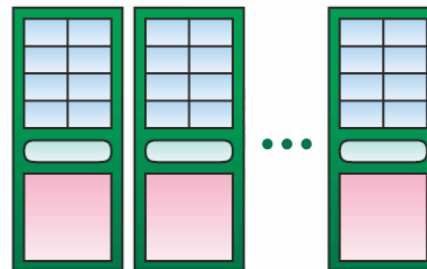
GPU



Thread Processor



Multi-processor



Device

CPU



Scalar SSE



Vector SSE



Multi-core

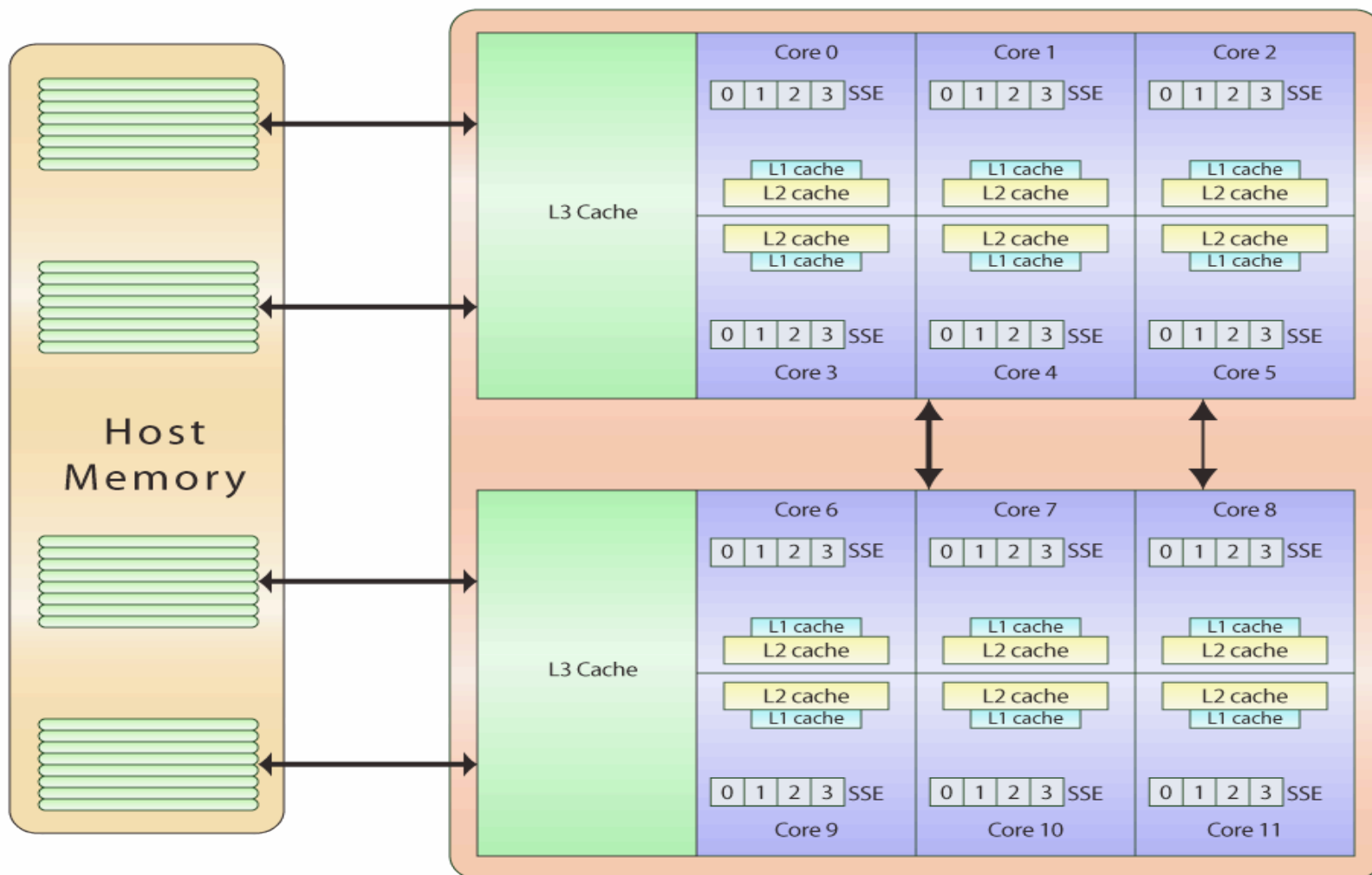
Optimized CUDA C for Multi-core x86

- ❑ Process CUDA C as a native parallel programming language for multi-core x86
- ❑ Inline Device kernel functions, translate chevron syntax to parallel/vector loops, use multiple cores and SSE/AVX instructions
- ❑ Execute each CUDA thread block using a single host core, eliminate synchronization where possible
- ❑ Host Code: all PGI optimizations for Intel/AMD host code will be supported
- ❑ Performance Goal: Well-structured CUDA C for multi-core x86 programs approach the efficiency of the same algorithm written in OpenMP



- ❑ **NVIDIA TESLA C1060/C2050**
 - Lots of available performance ~1 TFlops peak SP
 - Programming is a challenge
 - Getting high performance is lots of work
- ❑ **NVIDIA CUDA programming model and C for CUDA simplify GPGPU programming**
 - Much easier than OpenGL/DirectX, still challenging
- ❑ **PGI's CUDA Fortran provides an a Fortran based analog to CUDA C**
- ❑ **PGI's Accelerator Directive compilers for C and Fortran provide a higher level, OpenMP style of programming NVIDIA GPU's.**

AMD "Magny-Cours"



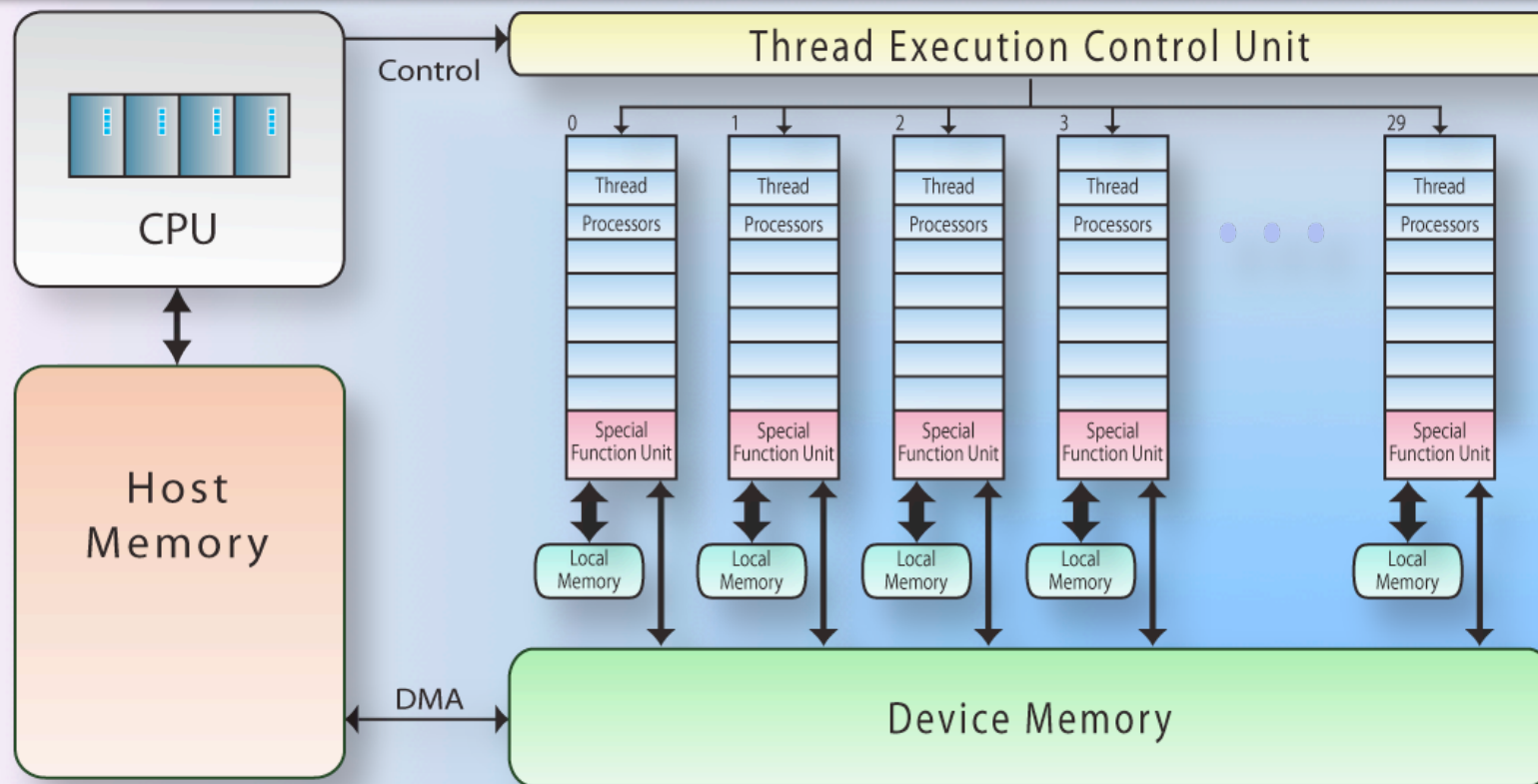
©2010 The Portland Group, Inc.



The Portland Group®

Emerging Cluster Node Architecture

Commodity Multicore x86 + Commodity Manycore GPUs



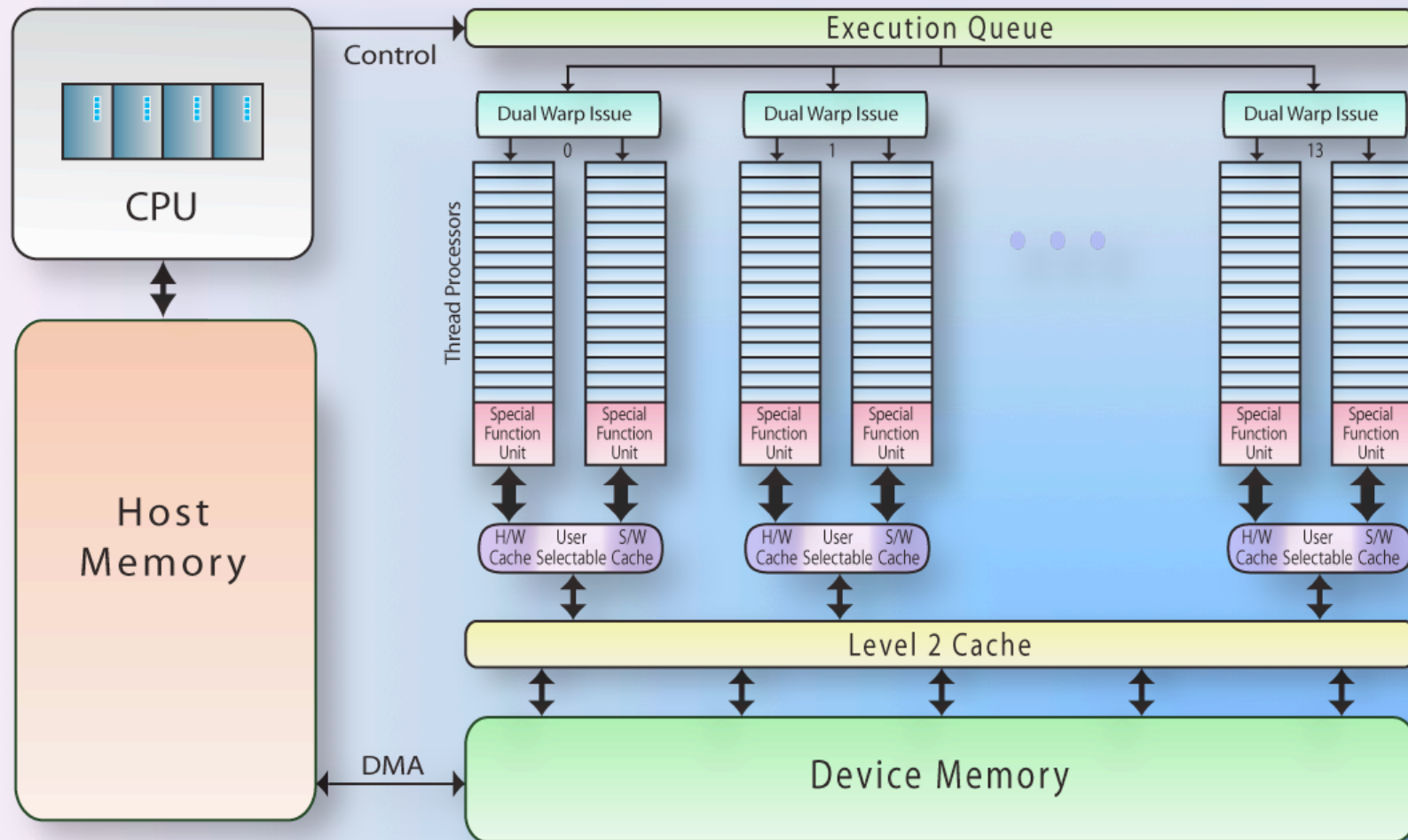
©2010 The Portland Group, Inc.

4 – 48 CPU Cores

240 – 1920 GPU/Accelerator Cores

The Portland Group®

Abstracted x64+Fermi Architecture



©2010 The Portland Group, Inc.

1-57

CUDA Fortran VADD Host Code

```
subroutine vadd( A, B, C )  
  use cudafor  
  use kmod  
  real(4), dimension(:) :: A, B  
  real(4), pinned, dimension(:) :: C  
  real(4), device, allocatable:: Ad(:), Bd(:), Cd(:)  
  integer :: N  
  N = size( A, 1 )  
  allocate( Ad(N), Bd(N), Cd(N) )  
  Ad = A(1:N)  
  Bd = B(1:N)  
  call vaddkernel<<<(N+31)/32,32>>>( Ad, Bd, Cd, N )  
  C(1:N) = Cd  
  deallocate( Ad, Bd, Cd )  
end subroutine
```

CUDA Fortran VADD Device Code

```
module kmod
  use cudafor
contains
  attributes(global) subroutine vaddkernel(A,B,C,N)
    real(4), device :: A(N), B(N), C(N)
    integer, value :: N
    integer :: i
    i = (blockidx%x-1)*32 + threadidx%x
    if( i <= N ) C(i) = A(i) + B(I)
  end subroutine
end module
```


3 Aspects of GPU Programming

1. Split code between Host and GPU

- CUDA and OpenCL – function level, done manually by the programmer
- Modern Compilers – can do this just as well as you can, and a lot faster, and enable offloading of regions within functions

2. Manage data allocation/movement between Host and Device

- CUDA and OpenCL – do this manually with API calls, one or more per argument to the device kernel, host code nearly unrecognizable compared to original
- Modern Compilers – can do this almost as well you can, user-driven tuning is required, but can and should be quick and easy

3. Tune Device Kernels

- CUDA and OpenCL – this step is both time-consuming and difficult; must optimize grid/thread geometry, optimize memory placement/accesses, etc
- Modern Compilers – can help a little here and make the code portable, but this step is probably always going to be *hard*

Explicit programming (CUDA) vs. implicit programming (directives)

CUDA:

- + Good performance with hand tuned kernels
- + Incremental porting to GPU
- not portable to non-CUDA platforms
- requires maintaining two sets of code

Directives:

- + Good performance possible
- + Incremental porting to GPU
- + portable to non-CUDA platforms including X64
- + requires only a single code source
- Obscurity in what the compiler is actually doing
- "Best practices" not clearly established – more data from user, vendor, and platform needed

PGI Accelerator Programming Model

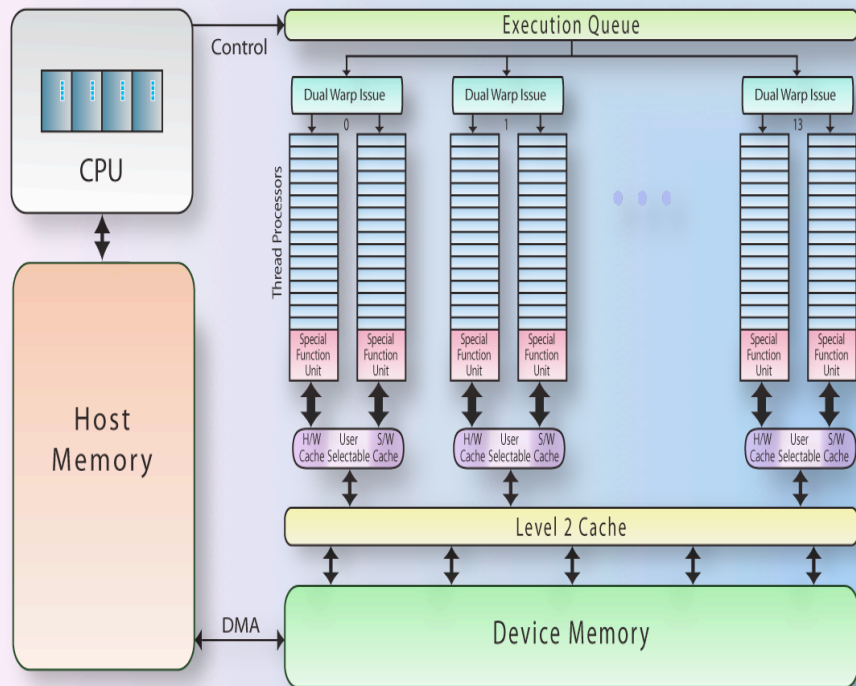
- ❑ Built on lessons from 30 years of experience with vector machines and 20 years of experience with SMP programming
- ❑ Directives to offload compute kernels to a GPU, manage data movement between host and GPU, map loop parallelism onto a GPU
- ❑ Fortran 2003 and C99 today, eventually C++
- ❑ Programs remain 100% standard compliant and portable to other compilers and HW
- ❑ Incremental porting/tuning of applications to x64+GPU
- ❑ Designed to enable development of applications that are performance portable to multiple types of accelerators

Basic C code - Matrix Multiply for x64 Single-core

```
void
computeMM(float C[][WB], float A[][WA], float B[][WB], int hA, int wA, int wB)
{
    int i, j, k;

    for (i = 0; i < hA; ++i) {
        for (j = 0; j < wB; ++j) {
            C[i][j] = 0.0;
        }
    }
    for (i = 0; i < hA; ++i) {
        for (k = 0; k < wA; ++k) {
            for (j = 0; j < wB; ++j) {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

NVIDIA CUDA C



©2010 The Portland Group, Inc.

Host Code

```
cudaMalloc(&A, bytes);
cudaMemcpy(A, data, bytes);
...
sgemm<<<dim3(m/16,n/16),dim3(16,16)>>>
    (A,la,B,lb,C,lc);
...
```

```
__global__ void sgemm( float *A, int la,
    float* B, int lb, float* C, int lc )
{
    int tx=threadIdx.x, ty=threadIdx.y;
    int i = blockIdx.x*16+tx;
    int j = blockIdx.y*16+ty;
    float Cij = C[i+j*lc];
    __shared__ float Ab[16][16];
    __shared__ float Bb[16][16];
    for(int kb=0; kb<lc; kb+=16){
        Ab[tx][ty] = A[i+la*(kb+ty)];
        Bb[tx][ty] = B[kb+tx+lb*(j)];
        __syncthreads();
        for(int k=0; k<16; ++k)
            Cij += Ab[tx][k]*Bb[k][ty];
        __syncthreads();
    }
    C[i+j*lc] = Cij;
}
```

GPU Code

PGI Accelerator Program Execution Model

□ Host

- executes most of the program
- allocates accelerator memory
- initiates data copy from host memory to accelerator
- sends kernel code to accelerator
- queues kernels for execution on accelerator
- waits for kernel completion
- initiates data copy from accelerator to host memory
- deallocates accelerator memory

□ Accelerator

- executes kernels, one after another
- concurrently, may transfer data between host and accelerator

Accelerating an Application

- Given that the app meets the constraints discussed, simply surround region to be accelerated with

```
!$acc region  
  <code loops>  
!$acc end region
```

- Compile for GPU

```
pgfortran -fast  
          -Minfo=accel  
          -ta=nvidia
```

Produces accelerated
kernels with correct
data movement

Compiler feedback
important for tuning

PGI Directive-based Matrix Multiply for x64+GPU

C code

```
void
computeMM(float C[][WB], float A[][WA],
          float B[][WB], int hA,
          int wA, int wB)
{
    int i, j, k;

    #pragma acc region
    {
        for (i = 0; i < hA; ++i) {
            for (j = 0; j < wB; ++j) {
                C[i][j] = 0.0;
            }
            for (k = 0; k < wA; ++k) {
                for (j = 0; j < wB; ++j) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
    }
}
```

- ❑ PGI 2010 automatically generates code for NVIDIA GPUs
- ❑ Generated code takes into account corner cases
- ❑ Block dimension chosen by the compiler is 16x16 threads
- ❑ Each thread of a given block computes one point in 'C' output matrix
- ❑ No use of shared memory for A & B accesses
- ❑ Code needs to be structured to enable « cache » accesses to A & B
- ❑ Single binary for both optimized versions for multicore and GPU

```

void saxpy (float a,
float *restrict x,
float *restrict y, int n){
#pragma acc region
{
    for (int i=1; i<n; i++)
        x[i] = a*x[i] + y[i];
}
}

```

compile

**PGI Accelerator
Compilers**

pgcc -ta=nvidia

GPU/Accelerator Code

Host x86 Code

```

saxpy:
...
movl    (%rbx), %eax
movl    %eax, -4(%rbp)
call    __pg_cu_init
...
call    __pg_cu_alloc
...
call    __pg_cu_uploadp
...
call    __pg_cu_paramset
...
call    __pg_cu_launch
...
Call    __pg_cu_downloadp
...

```

+

```

static __constant__ struct{
    int tc1;
    float* _y;
    float* _x;
    float _a;
}a2;

extern "C" __global__ void
pgi_kernel_2() {
    int i1, ils, ibx, itx;
    ibx = blockIdx.x;
    itx = threadIdx.x;
    for( ils = ibx*256; ils < a2.tc1; ils += gridDim.x*256 ){
        i1 = itx + ils;
        if( i1 < a2.tc1 ){
            a2._x[i1] = (a2._y[i1]+(a2._x[i1]*a2._a));
        }
    }
}

```

link

**Unified HPC
Application**

execute

... with no change to existing makefiles, scripts,
programming environment, etc

Refinements: Loop Schedules

```
Accelerator kernel generated  
26, #pragma acc for parallel, vector(16)  
27, #pragma acc for parallel, vector(16)
```

- ❑ vector loops correspond to threadidx indices
- ❑ parallel loops correspond to blockidx indices
- ❑ this schedule has a CUDA schedule:

```
<<< dim3(ceil(N/16),ceil(M/16)),dim3(16,16) >>>
```
- ❑ Compiler strip-mines to protect against very long loop limits, generates clean-up code for arbitrary loop bounds, etc
- ❑ Syntax supports any legal CUDA schedule

Refinements: Data Motion

```
!$acc region copyin(a(1:m,1:n)), copyout r
  do j = 2,n-1      ! Update interior points
    do i = 2,m-1
      r(i,j) = a(i,j) * 2.0
    enddo
  enddo
!$acc end region
```

- ❑ `copyin` changes default copyin procedure
- ❑ `copyout` changes default copyout procedure
- ❑ By default, compiler moves only data that is used
- ❑ E.g. for computing on non-halo regions, it is more efficient to move entire array rather than have the compiler generate a move for each column/row

Refinements: Leaving Data on GPU

```
Subroutine magma (A, B)
  real(4), dimension(:, :) :: A, B
  !$acc reflected (A,B)
```

- ❑ `reflected` is a new directive in PGI 11.0
- ❑ `reflected` requires visibility of the caller (module or interface block)
- ❑ Instructs the compiler that the data is already on the GPU
- ❑ Starts to help programmer work around the “no stack pointer” issue
- ❑ Replaces trying to get the compiler to inline called routines

Compiler-to-User Feedback

```
% pgfortran -fast -ta=nvidia -Minfo=accel mm.f
...
62, Loop is parallelizable
64, Loop carried dependence of 'C' prevents parallelization
    Loop carried backward dependence of 'C' prevents vectorization
66, Loop is parallelizable
    Accelerator kernel generated
62, #pragma acc for parallel, vector(16) /* blockIdx.y threadIdx.y */
64, #pragma acc for seq(16)
    Cached references to size [16x16] block of 'A'
    Cached references to size [16x16] block of 'B'
66, #pragma acc for parallel, vector(16) /* blockIdx.x threadIdx.x */
    Using register for 'C'
    CC 1.3 : 27 registers; 2264 shared, 24 constant,
            0 local memory bytes; 50% occupancy
...
```

Optional Region Clauses for Tuning Data Allocation and Movement

Clause	Scope
if (<i>cond</i>)	region
copy (<i>list</i>)	region, declaration
copyin (<i>list</i>)	region, declaration
copyout (<i>list</i>)	region, declaration
local (<i>list</i>)	region, declaration
mirror (<i>list</i>)	region, declaration (Fortran)
reflected (<i>list</i>)	declaration (Fortran)

See www.pgroup.com/accelerate for a complete specification of the PGI Accelerator programming model and directives

Optional Loop Directive Clauses for Tuning Kernel Schedules

Clause	Scope
host [(width)]	loop
parallel [(width)]	loop
seq [(width)]	loop
vector [(width)]	loop
private (list)	loop
kernel	loop
unroll (width)	loop
cache (list)	loop

See www.pgroup.com/accelerate for a complete specification of the PGI Accelerator programming model and directives

Device-Resident Data Example

```
change = tolerance + 1.0
!$acc data region local(newa(1:m,1:n)) &
    copy(a(1:m,1:n))
do while(change > tolerance)
    change = 0
!$acc region
    do i = 2, m-1
    do j = 2, n-1
        newa(i,j) = w0 * a(i,j) + &
            w1 * (a(i-1,j) + a(i,j-1) + &
                a(i+1,j) + a(i,j+1)) + &
            w2 * (a(i-1,j-1) + a(i-1,j+1) + &
                a(i+1,j-1) + a(i+1,j+1))
        change = max(change,abs(newa(i,j)-a(i,j)))
    enddo
    enddo
    a(2:m-1,2:n-1) = newa(2:m-1,2:n-1)
!$acc end region
    enddo
!$acc end data region
```

PGI Accelerator vs CUDA

- The **PGI Accelerator** programming model is a high-level *implicit* programming model for x64+GPU systems, similar to OpenMP for Multi-core x64:
 - Offload compute-intensive loops and code regions using simple compiler directives
 - Directives are Fortran comments and C pragmas, programs remain 100% standard-compliant and portable
 - Makes GPGPU programming and optimization incremental and accessible to application domain experts
 - Supported in both the PGI F2003 and PGCC C99 compilers

Reference Materials

- ❑ **PGI Accelerator programming model** – supported for x64+NVIDIA targets in the PGI Fortran 95/03 and C99 compilers
 - http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.3.pdf
- ❑ **CUDA Fortran** – supported on NVIDIA GPUs in PGI Fortran 95/03 compiler
 - <http://www.pgroup.com/lit/whitepapers/pgicudaforug.pdf>
- ❑ **Understanding the CUDA Data Parallel Threading Model**
 - <http://www.pgroup.com/lit/articles/insider/v2n1a5.htm>

Copyright Notice

© Contents copyright 2010, The Portland Group, Inc. This material may not be reproduced in any manner without the expressed written permission of The Portland Group.

PGFORTRAN, PGF95, PGI Accelerator and PGI Unified Binary are trademarks; and PGI, PGCC, PGC++, PGI Visual Fortran, PVF, PGI CDK, Cluster Development Kit, PGPROF, PGDBG, and The Portland Group are registered trademarks of The Portland Group Incorporated. Other brands and names are the property of their respective owners.

Parallel C code - Matrix Multiply for a Multi-core x64 Host

C code

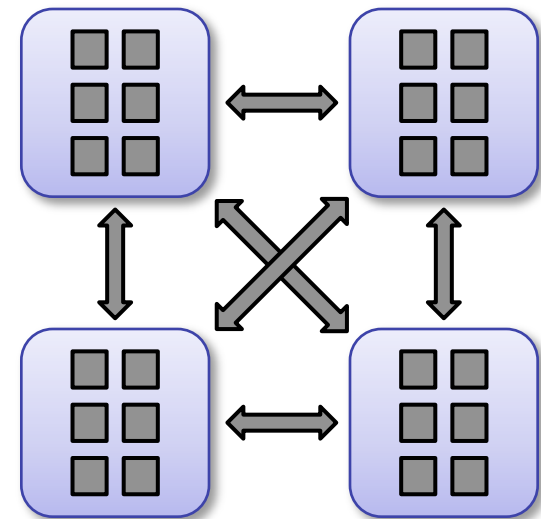
```
void
computeMM(float C[][WB], float A[][WA],
          float B[][WB], int hA, int wA,
          int wB)
{
    int i, j, k;
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < hA; ++i) {
            for (j = 0; j < wB; ++j) {
                C[i][j] = 0.0;
            }
        }
        #pragma omp for
        for (i = 0; i < hA; ++i) {
            for (k = 0; k < wA; ++k) {
                for (j = 0; j < wB; ++j) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
    }
}
```

PGCC
+
Vectorization
+
IPA
+
OpenMP



i-loop parallelized
+
k-loop vectorized

Parallel/Vector SSE code on Multi-core x64 SMP node

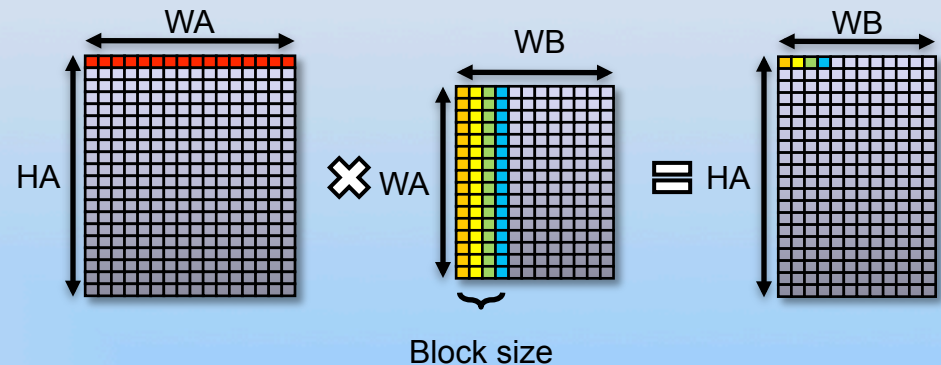


Basic CUDA C Matrix Multiply Kernel for an NVIDIA GPU

CUDA-C code

```
extern "C" __global__ void
mmkernel(float* A, float* B, float* C,
         int wA, int wB)
{
    int i = blockIdx.y;
    int j = blockIdx.x*64+threadIdx.x;

    float sum = 0.0;
    for( int k = 0; k < wA; ++k )
        sum += A[wA*i+k] * B[k*wB+j];
    C[i*wB+j] = sum;
}
```



- ❑ Here block size is equal to 64
- ❑ Each thread of a given block computes one point in 'C' output matrix
- ❑ Each thread of a given block reads same line of 'A' matrix
- ❑ Each thread of a given block read a different column from 'B' matrix

Optimized CUDA-C code

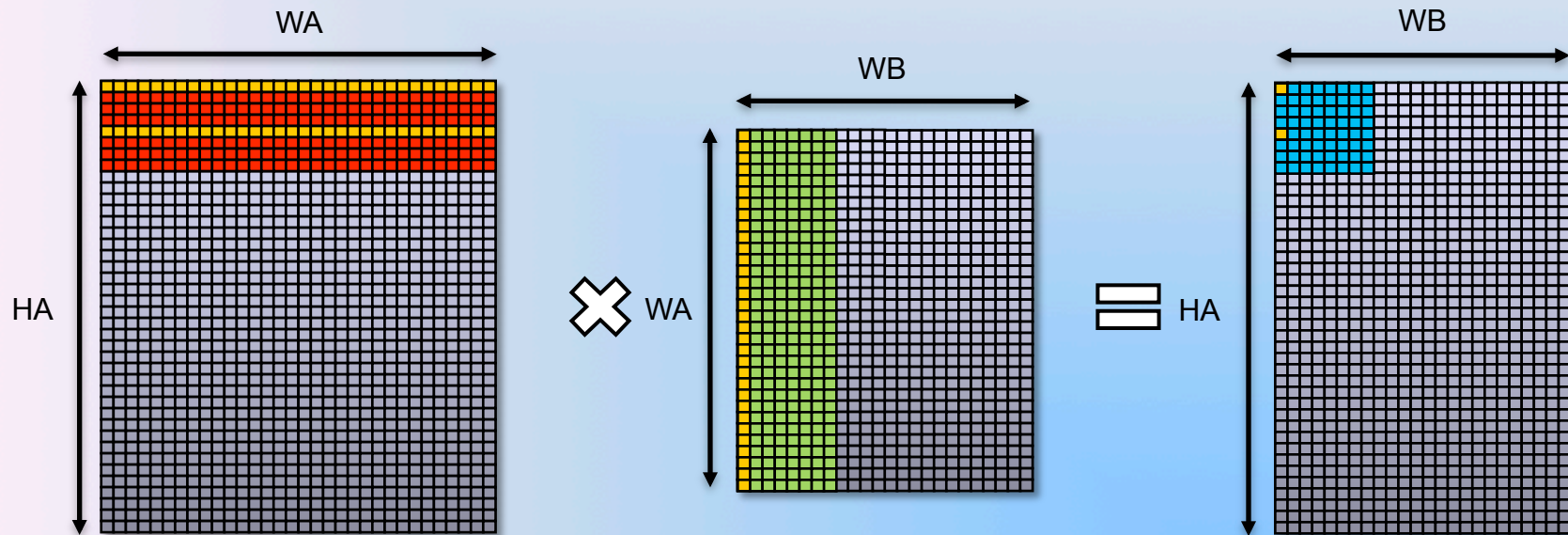
```
extern "C" __global__ void
c_mmul_kernel( float* c, float* a, float* b,
               int WA, int WB, int WC )
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int is = blockIdx.x*16, i = is + tx;
    int js = blockIdx.y*16, j = js + ty;
    __shared__ float at[16][16], bt[16][16];

    float sum0 = 0.0, sum1 = 0.0;
    for (int ks = 0; ks < WA; ks += 16){
        at[ty][tx] = a[ks+tx+WA*(ty+js)];
        at[ty+8][tx] = a[ks+tx+WA*(ty+js+8)];
        bt[ty][tx] = b[is+tx+WB*(ty+ks)];
        bt[ty+8][tx] = b[is+tx+WB*(ty+ks+8)];
        __syncthreads();
        for (int k = 0; k < 16; ++k)
            sum0 += at[ty][k]*bt[k][tx];
        for (int k = 0; k < 16; ++k)
            sum1 += at[ty+8][k]*bt[k][tx];
        __syncthreads();
    }
    c[i+WB*j] = sum0;
    c[i+WB*(j+8)] = sum1;
}
```

- ❑ Here thread block size is equal to 128
- ❑ Each thread of a given block computes 2 points in 'C' output matrix
- ❑ Computation performed on 16x16 data tiles of elements from 'A' and 'B'
- ❑ Each thread of a given block reads 2 elements of 'A' and 2 elements of 'B' and stores them in **shared** memory
- ❑ Each thread of a given block waits for other threads of the same block to complete caching of 'A' and 'B' into **shared** memory before starting any computation
- ❑ The j-loop has been unrolled by 2 to take advantage of re-using same 'at' element for different computations
- ❑ The inner k-loops will be completely unrolled (16x) by nvcc
- ❑ Each thread of a given block waits for other threads to complete computation before caching next 16x16 tiles of 'A' and 'B' matrices

Optimized CUDA C Matrix Multiply Kernel principle



- ❑ Execution of one block of threads
- ❑ Simplified example with block size equals to 8×4 instead of 16×8
- ❑ What about corner cases, i.e. matrix size not multiple of block dimension?

Host-side CUDA C Matrix Multiply GPU Control Code

```
cudaMalloc( &ap, memsizeA );  
cudaMalloc( &bp, memsizeB );  
cudaMalloc( &cp, memsizeC );
```

```
cudaMemcpy( ap, a, memsizeA, cudaMemcpyHostToDevice );  
cudaMemcpy( bp, b, memsizeB, cudaMemcpyHostToDevice );  
cudaMemcpy( cp, c, memsizeC, cudaMemcpyHostToDevice );
```

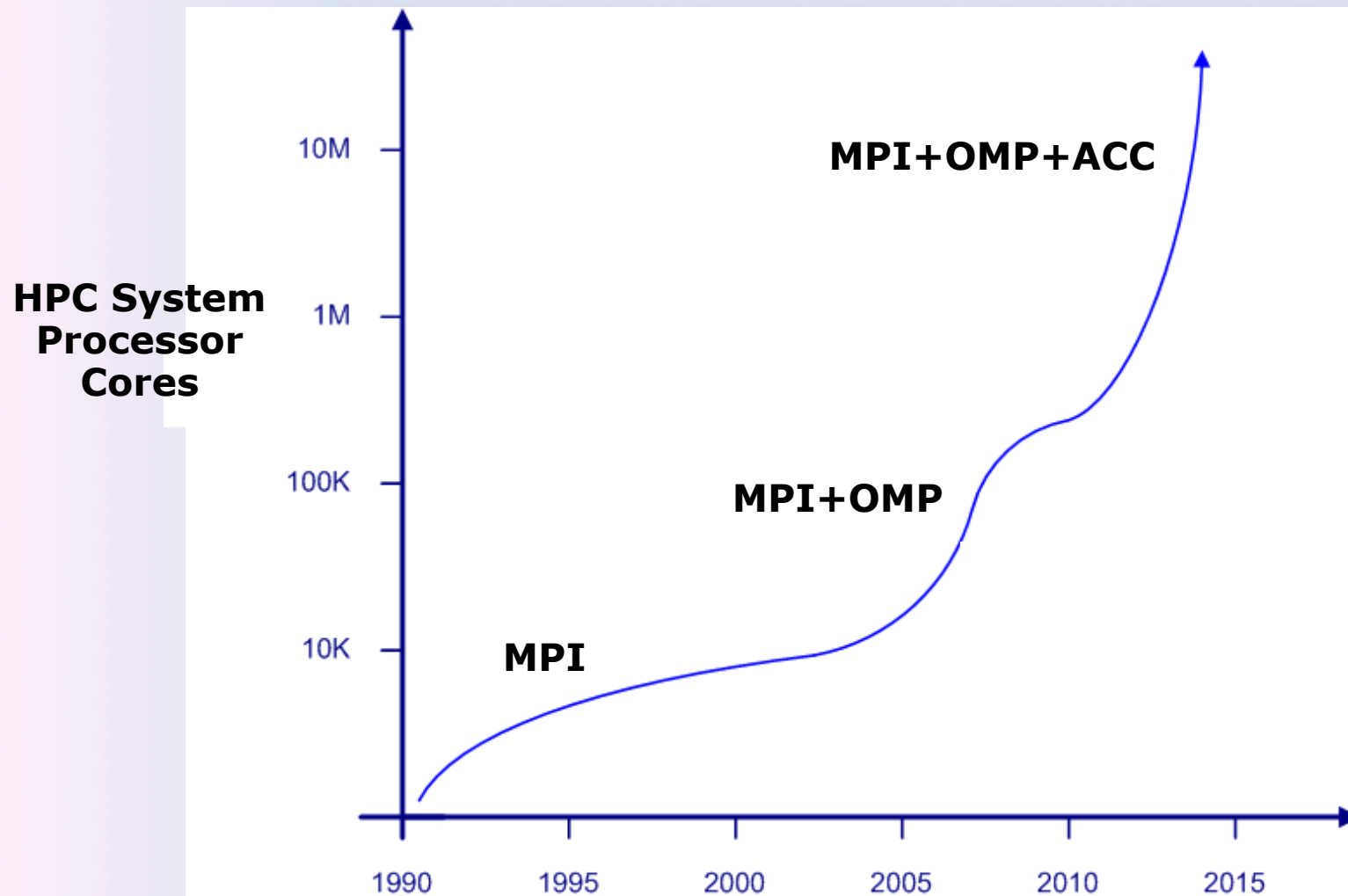
```
dim3 threads( 16, 8 );  
dim3 blocks( hA/16, wB/16 );  
c_mmul_kernel <<<blocks,threads>>>(ap, bp, cp,  
                                     wA, wB, wC);
```

```
cudaMemcpy( C, cp, memsizeC, cudaMemcpyDeviceToHost );
```

```
cudaFree( ap );  
cudaFree( bp );  
cudaFree( cp );
```

Compilers & Programming Models Must Evolve for Each New Generation of HPC Hardware

Expect 20M Core systems in the Next Few Years






- ❑ **NVIDIA TESLA C1060 and C2050 (Fermi)**
 - Lots of available performance 1 - 2 TFlops peak SP
 - Programming is a challenge
 - Getting high performance is lots of work
- ❑ **NVIDIA CUDA programming model simplifies GPGPU programming**
 - CUDA C much easier than OpenGL, still challenging
 - PGI CUDA Fortran provides a Fortran solution
- ❑ **A PGI Goal: do for GPU programming what OpenMP did for Posix Threads, make it easily approachable by application domain experts**

PGI Directive-based Matrix Multiply for Multi-core x64+GPU

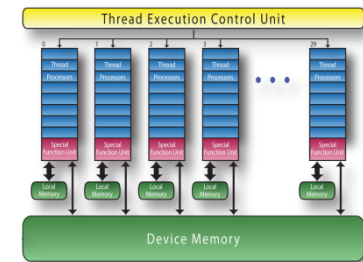
C code

```
void
computeMM(float C[][WB], float A[][WA],
          float B[][WB], int hA, int wA,
          int wB)
{
    int i, j, k;
    #pragma acc region
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < hA; ++i) {
            for (j = 0; j < wB; ++j) {
                C[i][j] = 0.0;
            }
        }
        #pragma omp for
        for (i = 0; i < hA; ++i) {
            for (k = 0; k < wA; ++k) {
                for (j = 0; j < wB; ++j) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
    }
}
```

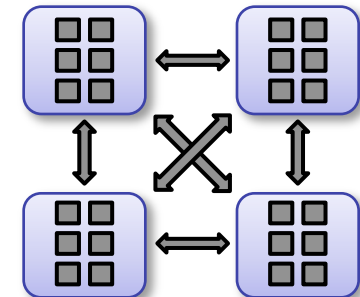
PGCC
+
PGI Accelerator
+
IPA
+
Vectorization
+
OpenMP
+
PGI Unified Binary



Autoparallelized Multi-Dimension code on GPU



Parallel/Vector SSE code on Multi-core x64 SMP node



Maybe OpenCL is Easier?

OpenCL code

```
__kernel void
matrixMul( __global float* C,
           __global float* A, __global float* B,
           __local float* As, __local float* Bs)
{
    int bx = get_group_id(0);
    int tx = get_local_id(0);
    int by = get_group_id(1);
    int ty = get_local_id(1);
    int aEnd  = WA * BK_SZ * by + WA - 1;
    int a = WA*BK_SZ*by ;
    int b = BK_SZ * bx
    float Csub = 0.0f;

    for (;a <= aEnd; a += BK_SZ, b += BK_SZ*WB) {
        As[tx + ty * BK_SZ]=A[a + WA * ty + tx];
        Bs[tx + ty * BK_SZ]=B[b + WB * ty + tx];
        barrier(CLK_LOCAL_MEM_FENCE);
        for (int k = 0; k < BK_SZ; ++k)
            Csub+=As[k+ty*BK_SZ]*Bs[tx+k*BK_SZ];
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    int c = get_global_id(1)*get_global_size(0);
    c = c + get_global_id(0);
    C[c] = Csub;
}
```

OpenCL Matrix Multiply Kernel for GPU

- ❑ Here *WorkGroup* size is 16x16
- ❑ Each *WorkItem* of a given *WorkGroup* computes one point in 'C' output matrix
- ❑ Computation is performed in sub-arrays of 16x16 elements of 'A' & 'B' matrices
- ❑ Each *WorkItem* of a given *WorkGroup* reads one element of 'A' & 'B' matrices and stores them in **local** memory
- ❑ Each *WorkItem* of a given *WorkGroup* waits for other *WorkItems* of same *WorkGroup* to complete caching of 'A' & 'B' matrices into **local** memory before starting any computation
- ❑ Each *WorkItem* of a given *WorkGroup* waits for other *WorkItems* to complete computation before caching next 16x16 sub-array of 'A' & 'B' matrices

OpenCL Host-side C Control Code

Step1/3: Platform capability query, context creation, command queue creation

```
ciErrNum = oclGetPlatformID(&cpPlatform);
...
ciErrNum = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 0, NULL, &ciDeviceCount);
...
ciErrNum = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, ciDeviceCount, cdDevices, NULL);
...
cxGPUContext = clCreateContext(0, ciDeviceCount, cdDevices, NULL, NULL, &ciErrNum);
...
    ciErrNum |= clGetContextInfo(cxGPUContext, CL_CONTEXT_DEVICES, 0, NULL, &nDeviceBytes);
ciDeviceCount = (cl_uint)nDeviceBytes/sizeof(cl_device_id);

clGetContextInfo(cxGPUContext, CL_CONTEXT_DEVICES, 0, NULL, &szParmDataBytes);

cdDevices = (cl_device_id*) malloc(szParmDataBytes);

clGetContextInfo(cxGPUContext, CL_CONTEXT_DEVICES, szParmDataBytes, cdDevices, NULL);

cl_device_id device = cdDevices[0];
commandQueue = clCreateCommandQueue(cxGPUContext, device, 0, &ciErrNum);
...
```


OpenCL Host-side C Control Code

Step2/3: Program Building, Kernel creation, Resources allocation, argument passing, kernel enqueueing

```
char *source = loadProgSource("matrixMul.cl", header, &program_length);
...
cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char **)&source,
                                     &program_length, &ciErrNum);
...
ciErrNum = clBuildProgram(cpProgram, 0, NULL, "-cl-mad-enable", NULL, NULL);
...
multiplicationKernel = clCreateKernel(cpProgram, "matrixMul", &ciErrNum);
...
d_A = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                    mem_size_A, &ciErrNum);
d_B = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                    mem_size_B, h_B_data, NULL);
d_C = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY, mem_size_C, NULL, NULL);

clSetKernelArg(multiplicationKernel, 0, sizeof(cl_mem), (void *) &d_C);
clSetKernelArg(multiplicationKernel, 1, sizeof(cl_mem), (void *) &d_A);
clSetKernelArg(multiplicationKernel, 2, sizeof(cl_mem), (void *) &d_B);
clSetKernelArg(multiplicationKernel, 3, sizeof(float) * BK_SZ * BK_SZ, 0);
clSetKernelArg(multiplicationKernel, 4, sizeof(float) * BK_SZ * BK_SZ, 0);
...
size_t localWS[] = {BLOCK_SIZE, BLOCK_SIZE};
size_t globalWS[] = {WC, HA};

clEnqueueNDRangeKernel(commandQueue, multiplicationKernel, 2, 0, globalWS, localWS,
                      0, NULL, &GPUExecution);
```

OpenCL Host-side C Control Code

Step3/3: Kernel execution, results copy, resources deallocation

```
...  
    clFinish(commandQueue);  
  
    clEnqueueReadBuffer(commandQueue, d_C, CL_FALSE, 0, mem_size_C,  
                        h_C, 0, NULL, &GPUDone);  
    clWaitForEvents(ciDeviceCount, GPUDone);  
  
    clReleaseMemObject(d_A);  
    clReleaseMemObject(d_C);  
    clReleaseMemObject(d_B);  
        clReleaseEvent(GPUExecution);  
        clReleaseEvent(GPUDone);  
  
    clReleaseKernel( multiplicationKernel );  
    clReleaseCommandQueue( commandQueue );  
    clReleaseProgram(cpProgram);  
    ciErrNum = clReleaseContext(cxGPUContext);
```

Session Edit View Bookmarks Settings Help

```
grandcanyon:% pgf95 -Mmpi=mpich -fast -c -Minfo=all,intensity
baroclinic.fppized.f90
init_baroclinic:
140, Loop not
```

File Settings Sort Search Help

PGDBG - The Portland Group

File Edit View Data Debug Help

Current Thread

Thread 0

Source Disassembly Mixed

Line No. Event PC s:/demos/POP_WS_Windows/de

526 if (lpressure_avg .and. le

527 call update_ghost_cells

528

529 call update_ghost_cells

530

531 endif

532

533

534

535

536

537

538

539 !OMP PARALLEL DO PRIVATE(i

540 !OMP WORK, FX, FY, WORK1, t

541

542 do iblock = 1,nblocks_clin

543

544 this_block = get_block(i

545

546

547

548 initialize arrays for verti

549

550

551

#0 baroclinic_driver line: 539 in "baroclinic.

Command Events Groups

[[8] New Thread)

[[9] New Thread)

[[10] New Thread)

[[11] New Thread)

[[12] New Thread)

[[13] New Thread)

[[14] New Thread)

[[15] New Thread)

[0] Breakpoint at 0x140013E4B, function baroclinic_dr

#539: !OMP PARALLEL DO PRIVATE(iblock,k,kml,

pgdbg [all] 0> |

Stopped at line 539 (address 0x140013e4b) in file s:/demos/POP_WS_Windows

Find:

pgprof-4T-time.out [0]

mm_fv_update_nonb...

HotSpot: Seconds

Line Source Seconds

1123 al->qxz += k*(*nodelistt)[inode].q012;

1124 #endif

1125

1126 } /* if innode > 0 end if */

1127

1128

1129

1130

1131

1132 for(i=0; i< nng0; i++)

1133 {

1134 a2 = (*atomall)[natoms*o+i];

1135 /* add the new components */

1136 #ifdef _OPENMP

1137 omp set lock(&(a2->lock));

Sorted By Line

Line-level information for line 1132

1. Intensity = 102.00

- Compute Intensity is a ratio of computation to data movement, and is one measure to identify candidates for offloading to a GPU.

- Optimization Hint: If the loop or routine has compute intensity greater than 1, consider accelerating it using the PGI Accelerator Model or CUDA Fortran.

2. Loop not vectorized/parallelized: contains call

- Optimization Hint: Try compiling with inlining options or manually splitting the loop

- Note that I/O statements, intrinsics, and overloaded operators may be implemented as procedure calls

Information about how file ../src/rectmm.c was compiled

Parallelism Histogram Compiler Feedback System Configuration Accelerator Performance

Profiled: ./ammp on Wed Nov 03 14:56:56 PDT 2010

Profile: ./pgprof-4T-time.out

Grid options

Grandcanyon

HPC Hardware Trends

Today: Clusters of Multicore x86

Tomorrow? Clusters of Multicore x86 + Accelerators

**Top
500**

